SENTRY: LEARNING TO RECOVER FROM FAILURES IN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents often fail mid-task due to invalid tool calls, repeated actions, or poorly grounded reasoning, and learning from these failures is a natural path to reliability. We find that *how* failure knowledge reaches the agent matters as much as *what* it contains. Failure lessons are conditional: kept in the agent's context, they misfire when their failure is absent, and removing them from an evolving playbook improves performance. Runtime interventions, in contrast, act only when a failure occurs but do not learn from their own repairs. We argue that failure knowledge is conditional knowledge and should be conditionally exposed, and instantiate this principle in Sentry, a failure-management layer that runs alongside the task agent. When Sentry detects a failure, it retrieves matching lessons from an external playbook to guide recovery, verifies without access to task rewards whether the agent recovered, and stores a new lesson only if it did; the full playbook never enters the agent's context. Across multiple agentic benchmarks, Sentry outperforms the strongest runtime-intervention baseline on every benchmark, by 37% on average, and the strongest context-evolution baseline by 39% on the two benchmarks where both are evaluated; combining Sentry with context evolution yields further gains. Learned lessons transfer to held-out tasks, and controlled experiments show that exposing the full playbook to the agent lowers performance even when relevant lessons remain available on demand.

1 INTRODUCTION

Modern large language model (LLM) agents increasingly tackle interactive and multi-step tasks involving reasoning, tool use, and external environments (Yao et al., 2023; Schick et al., 2023; Yao et al., 2024; Merrill et al., 2026). Success often hinges less on getting every step right than on recovering when one goes wrong: an invalid tool call (Winston & Just, 2025), a repeated or stalled action, a poorly grounded decision (Zhu et al., 2025), or reasoning that does not match the action (Cemri et al., 2025). As tasks grow longer, failure recovery becomes central to agent reliability.

Two lines of work let agents learn from failures, and each gets only half the problem right. *Context-evolution* methods distill past trajectories into reflections, workflows, or playbooks reused on future tasks (Shinn et al., 2023; Wang et al., 2025; Ouyang et al., 2026; Zhang et al., 2026c). They learn across tasks but expose lessons too broadly: even retrieval-based memories (Zhao et al., 2023; Ouyang et al., 2026) select lessons once at task start, before any failure, and keep them in context for the whole trajectory. *Runtime-intervention* methods monitor the trajectory and inject guidance when a failure is detected (Koohestani, 2025; Zhang et al., 2026a; Nanda et al., 2026; Mulian et al., 2026). They act selectively but do not learn: what resolved one failure does not inform the next, and a fixed rule that rescues one task can derail another (Figure 1b).

We find that *how* failure knowledge reaches the agent matters as much as *what* it contains. Failure lessons are conditional:"if repeated searches return nothing relevant, broaden the query" helps only when that failure is occurring. Kept in context, such lessons force the agent to judge at every step whether each applies, and it often judges wrong: in Figure 1a, an agent carrying recovery rules learned by ACE (Zhang et al., 2026c) finds a matching product but keeps comparing alternatives and never buys it. The effect is systematic. Removing failure lessons from ACE's playbook *improves* held-out performance, from 0.335 to 0.391 reward on WebShop (Yao et al., 2022) and from 0.562 to 0.643 completion on Mind2Web Replay (Deng et al., 2023). Conversely, adding a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Figure 1: **Two ways failure knowledge goes wrong** (Qwen3.5-9B on WebShop). (a) *Always exposed:* recovery rules learned by ACE stay in the agent's context; after finding a matching item, the agent keeps comparing alternatives and takes no action. (b) *Never learned:* AgentGuard's fixed no-progress warning resolves a search loop, but no lesson is stored; in a later task, the same warning fires after valid option selections, and the agent leaves the page and loses them.

fixed playbook to persistent context lowers WebShop reward from 0.364 to 0.296, even when the same lessons remain available on demand, mirroring capability erosion in self-evolving agents (Yu et al., 2026). Intervention shows the same pattern: intervening without checking whether it is needed drops WebShop reward below the unassisted agent's (0.136 vs. 0.170). These observations suggest a simple principle: **failure knowledge is conditional knowledge and should be conditionally exposed.** It should be *learned* across tasks, as in context evolution, but *surfaced* only on a matching failure, as in runtime intervention.

We instantiate this principle in ***Sentry***, a failure-management layer that runs alongside the task agent and learns from the recoveries it verifies. Sentry watches the agent's recent steps and returns non-executable actions for a schema-conforming retry. For behavioral failures, such as loops or reasoning not grounded in observations, Sentry diagnoses the failure, retrieves at most k matching lessons from an external playbook, and uses them to guide the agent's next step. It then checks whether the agent recovers, without access to task rewards, and writes a new lesson only if it does. The full playbook never enters the agent's context: the agent sees a handful of lessons, only while failing, and only for that failure. Because the problem stems from how knowledge enters the context rather than from a particular model's weaknesses, we expect it to matter more, not less, as memories grow and agents run longer.

Across WebShop, AppWorld (Trivedi et al., 2024), SWE-bench Lite (Jimenez et al., 2024), and Mind2Web Replay, Sentry outperforms the strongest runtime-intervention baseline on every benchmark, by 37% on average (up to 77%), and ACE, the strongest context-evolution baseline, by 39%. The gains are not just action repair: without the schema-conforming retry, Sentry still beats all runtime-intervention baselines on both ablated benchmarks. Sentry also complements context evolution: ACE+Sentry performs best wherever evaluated, suggesting that task- and failure-level knowledge can evolve together. Our contributions are:

- **A finding about failure knowledge.** We show that failure-specific knowledge is conditional, and that exposing it unconditionally hurts: always-on failure lessons and always-on intervention both lower agent performance in controlled comparisons (§2, §5.3).
- **Sentry, a method built on this finding.** Sentry keeps an evolving recovery playbook outside the agent's context, retrieves from it only when a matching failure is diagnosed mid-trajectory, and admits a lesson only after verifying, without reward access, that the recovery succeeded (§3).
- **Evidence across four benchmarks.** Beyond the main comparisons, held-out evaluation shows that learned recovery lessons transfer to unseen tasks; paired forks from 200 intervention points show that Sentry's guidance causally improves local recovery (33.5% $\rightarrow$ 60.5%); and human annotation agrees with Sentry's intervention and recovery judgments in 86.0% and 81.8% of cases (§5).

2 Background and Motivation

Methods that let LLM agents learn from execution failures differ along two axes: *whether* failure knowledge accumulates across tasks, and *what it is conditioned on* when it reaches the agent. We

examine both families along these axes; Figure 1 shows how each goes wrong on WebShop with Qwen3.5-9B (full trajectories in Appendix A.2; further related work in Appendix A.1).

Context evolution conditions on the task, but failures happen mid-trajectory. Reflexion stores natural-language reflections (Shinn et al., 2023), AWM induces reusable workflows (Wang et al., 2025), and ACE maintains an evolving playbook of advice (Zhang et al., 2026c); ExpeL and ReasoningBank retrieve a subset of lessons for each new task (Zhao et al., 2023; Ouyang et al., 2026). In all cases, lessons are selected before execution begins and stay in context throughout the task. This is the wrong conditioning variable for failure knowledge: whether "broaden the query after repeated empty searches" applies depends on the agent's state at a particular step, which is unknown when lessons are selected. Better task-level retrieval cannot fix this, and the agent is left to judge at every step which lessons apply. In Figure 1(a), three such rules keep the agent comparing alternatives after it has already found a matching product.

Runtime intervention conditions on the state, but cannot learn its own boundaries. AgentGuard warns the agent when fixed rules fire, e.g., on invalid actions or no-progress steps (Koohestani, 2025); AgentForesight raises early alarms from partial trajectories (Zhang et al., 2026a); Wink uses an LLM monitor to suggest course corrections (Nanda et al., 2026); and AgentFixer combines rule-based checks with LLM judges (Mulian et al., 2026). These methods condition on the right variable, the current trajectory, but their knowledge is fixed. Each intervention is written from scratch or from a hand-set rule, so nothing tells the system when an intervention helped, when it hurt, or how to tell the two apart next time. In Figure 1(b), "page unchanged" signals a genuine stall in one task and valid option selection in another; AgentGuard's rule cannot learn the difference.

Always-on failure knowledge costs more than it helps. To isolate the cost of exposure, we remove failure-mode lessons from ACE's playbook under the held-out protocol of Table 2 (memory initialized on 50 tasks, evaluated on 50 disjoint tasks; §4.3). Pruning improves ACE on both WebShop and Mind2Web Replay, our interactive adaptation of Mind2Web (Deng et al., 2023) (§1). On average, then, the harm these lessons cause when their failure is absent outweighs the help they give when it occurs. Pruning is a blunt fix, however: it discards that help along with the harm. The alternative is to keep the lessons but change *when* the agent sees them: with the same playbook, failure-triggered exposure outperforms persistent exposure (§5.3).

Takeaway. Failure knowledge should be *learned* across tasks, as in context evolution; *surfaced* only when a failure is diagnosed mid-trajectory, as in runtime intervention; and *admitted* only once its recovery is verified, since a lesson learned from a failed repair would misfire on future tasks too. Sentry (§3) is built around these three properties.

3 SENTRY: A FRAMEWORK FOR FAILURE RECOVERY IN LLM AGENTS

We introduce ***Sentry*** (Figure 2), a failure-management layer that runs alongside the task agent and handles failure knowledge along the three properties of §2: it *surfaces* recovery lessons only when it diagnoses a failure (§3.1, §3.2), *admits* a lesson only after verifying the recovery, and *learns* by accumulating verified lessons across tasks (§3.3). The complete playbook is maintained outside the agent's persistent background context; we refer to this as *isolation*. Isolation does not hide recovery knowledge from the agent: the few entries retrieved for a detected failure still enter the conversation during that intervention.

During execution, Sentry monitors the agent's recent trajectory. It routes invalid actions to *hard repair*, which asks the task agent for a valid replacement action, and routes progress and reasoning–grounding failures to *soft repair*, which combines the local diagnosis with relevant playbook entries to guide the agent's next step. After a soft repair, Sentry verifies whether the triggering failure has been resolved and adds only verified recovery lessons to the playbook for later interventions. Prompt templates are provided in Appendix A.8.3.

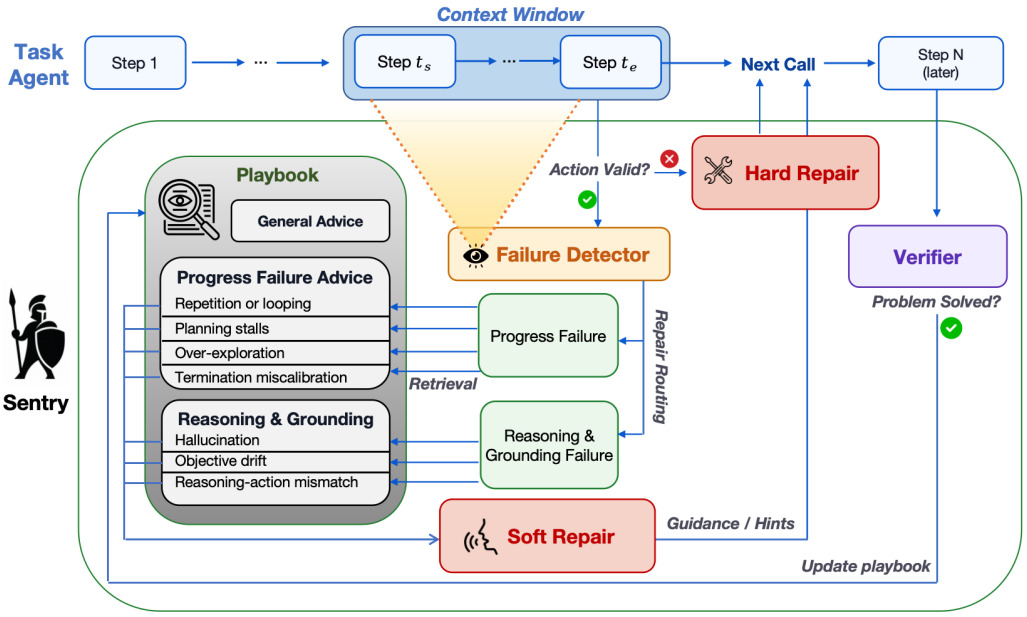


Figure 2: **Sentry's framework.** Sentry routes invalid actions to hard repair and progress or reasoning–grounding failures to soft repair with retrieved playbook guidance. Hard repairs are checked immediately; soft repairs undergo delayed recovery verification. Only verified soft-repair recoveries are added to the external playbook for use in future interventions.

3.1 Failure Detection and Routing

At each step, Sentry decides whether to intervene and, if so, which repair route to take. Given the task objective, it examines the current action and the agent's $W = 5$ most recent reasoning–action–observation cycles.

Sentry first checks whether the action is executable under the environment's required schema. If it is not, Sentry invokes *hard repair*: the task agent receives the invalid action, the relevant trajectory context, and the schema, and is instructed to output only a corrected action in the required format.

If the action is valid, Sentry checks the recent trajectory for two types of failures: *Progress Failures*, such as repetition or stalled execution, and *Reasoning & Grounding Failures*, such as unsupported claims or reasoning–action mismatch. For a detected failure, Sentry produces a diagnosis consisting of the failure type, fine-grained retrieval labels, and supporting local evidence, which then triggers *soft repair*. If no failure is detected, execution continues with no intervention and no playbook retrieval, so the agent sees no recovery knowledge while it is making progress. Full failure and retrieval-label definitions are provided in Appendix A.8.1.

3.2 Soft Repair with Playbook Retrieval

Each playbook entry is a record of a verified recovery: its failure type, retrieval labels, trigger pattern, and repair principle. Given a diagnosis, Sentry selects entries with the same failure type and at least one shared retrieval label, ranks them by label overlap, and retrieves at most $k = 5$. If no entry shares a retrieval label, Sentry falls back to the most recent entries of the same failure type. The procedure, including recency tie-breaking and label-diversity constraints, is provided in Appendix A.8.2.

Sentry then builds a *soft-repair* prompt from the retrieved lessons, the failure type, the supporting local evidence, and general repair instructions. The prompt guides the task agent to address the failure and choose a productive next step while staying on the original task objective. When the playbook is empty, soft repair relies on the general repair instructions alone; as verified rescues accumulate, retrieved lessons increasingly shape the guidance. Because retrieval is bounded, the playbook can grow across tasks while each intervention exposes the agent to at most k lessons.

3.3 Recovery verification and playbook update

Before any experience enters the playbook, Sentry verifies that the intervention resolved the detected failure. For hard repair, verification is immediate: the corrected action must satisfy the required schema and be accepted by the environment. For soft repair, Sentry examines the subsequent $H = 10$ reasoning–action–observation cycles and judges the failure resolved when the agent stops the diagnosed failure pattern and resumes task-relevant, observation-grounded progress. This judgment concerns recovery from the triggering failure only; the verifier has no access to benchmark rewards or final task-success labels.

Only soft-repair episodes judged resolved contribute new playbook entries; unresolved episodes produce none. For each resolved episode, Sentry summarizes the diagnosis, repair guidance, and observed recovery into a reusable record with the failure type, retrieval labels, trigger pattern, and repair principle. Verification thus controls *which* experiences enter the playbook, while failure-conditioned retrieval controls *when* they are activated. Self-evolution occurs only through playbook updates; the detection and retrieval procedures remain fixed. Appendix A.8.4 provides an example playbook produced by Sentry.

4 Experimental setup

4.1 Benchmarks and metrics

Benchmarks. We evaluate on WebShop and Mind2Web Replay (web agents), SWE-bench Lite (software engineering), and AppWorld (stateful app-based tool use) (Yao et al., 2022; Deng et al., 2023; Jimenez et al., 2024; Trivedi et al., 2024). Mind2Web Replay is our interactive adaptation of Multimodal-Mind2Web: after a rejected action, the agent remains at the reference state and may retry within its execution budget, so the benchmark measures feedback-driven recovery along trajectories. All methods share the same candidate-based interface and feedback; scores are not comparable to the official Mind2Web step-wise metrics (Appendix A.3.1). The runtime-intervention comparison uses 100 tasks on WebShop, Mind2Web Replay, and AppWorld, and 50 on SWE-bench Lite.

Metrics. We report average reward on WebShop; mean normalized trajectory completion, the fraction of reference steps completed in order, on Mind2Web Replay; issue-resolution accuracy under the official harness on SWE-bench Lite; and test pass rate, the percentage of checks passed per task, on AppWorld. Secondary metrics and intervention counts are in Appendices A.3.1 and A.5.1.

4.2 Baselines

The **Base agent** runs the model without runtime intervention or evolution. Runtime-intervention baselines are **AgentGuard** (Koohestani, 2025), **AgentFixer** (Mulian et al., 2026), **Wink** (Nanda et al., 2026), and **AgentForesight** (Zhang et al., 2026a); context-evolution baselines are **Reflexion** (Shinn et al., 2023), **ACE** (Zhang et al., 2026c), **AWM** (Wang et al., 2025), and **ReasoningBank** (Ouyang et al., 2026). We also evaluate **ACE+Sentry**, which adds Sentry's failure manager to ACE. Implementations are in Appendix A.3.2.

4.3 Implementation Details and Evaluation Protocols

By default, the task agent and all Sentry components use Qwen3.5-9B (Qwen Team, 2026); we also evaluate GPT-OSS-120B (OpenAI, 2025). Task sets are sampled once from the executable tasks of each official split and held fixed across five runs with randomized execution order; we report mean $\pm$ standard deviation over these runs. Decoding settings, output schemas, and task-selection details are in Appendices A.8.3 and A.4.

Evaluation protocols. The runtime-intervention comparison and component ablations use an *online-from-scratch* protocol: Sentry starts with an empty playbook and accumulates verified recoveries online. Context-evolution comparisons and the playbook exposure, transfer, and retrieval analyses use a *held-out* protocol on WebShop and Mind2Web Replay: each method initializes its memory on 50 tasks and is evaluated on 50 disjoint tasks, with set membership fixed across runs. During

Table 1: Sentry compared with runtime intervention baselines across four agentic benchmarks using Qwen3.5-9B. We report mean ± standard deviation over five repeated runs. Bold denotes the best measured result per column.

Method	WebShop avg. reward	AppWorld test pass (%)	SWE-bench Lite accuracy	Mind2Web Replay completion
Base agent	0.170 ± 0.016	25.59 ± 0.88	0.20 ± 0.01	0.420 ± 0.030
AgentGuard	0.258 ± 0.019	25.09 ± 0.91	0.22 ± 0.02	0.450 ± 0.019
AgentForesight	0.168 ± 0.011	24.17 ± 0.91	0.24 ± 0.01	0.430 ± 0.016
AgentFixer	0.171 ± 0.017	26.69 ± 0.80	0.22 ± 0.01	0.687 ± 0.027
Wink	0.264 ± 0.015	27.56 ± 0.98	0.18 ± 0.02	0.370 ± 0.020
Sentry	$\mathbf{0.468 \pm 0.017}$	$\mathbf{38.53 \pm 0.87}$	$\mathbf{0.30 \pm 0.01}$	$\mathbf{0.736 \pm 0.019}$

Table 2: **Comparison with context-evolution baselines on WebShop and Mind2Web Replay using Qwen3.5-9B.** Methods with memory initialize it on 50 tasks, then continue updating according to their respective procedures during evaluation on 50 disjoint tasks per benchmark (§4.3). Only held-out tasks contribute to the reported scores. Results are mean ± standard deviation over five runs.

Method	WebShop avg. reward	Mind2Web Replay completion
Base agent	0.186 ± 0.014	0.421 ± 0.026
Reflexion	0.295 ± 0.025	0.531 ± 0.023
ACE	0.335 ± 0.019	0.562 ± 0.020
AWM	0.270 ± 0.021	0.470 ± 0.018
ReasoningBank	0.250 ± 0.015	0.483 ± 0.020
Sentry	0.495 ± 0.017	0.729 ± 0.019
ACE+Sentry	$\mathbf{0.555 \pm 0.027}$	$\mathbf{0.789 \pm 0.022}$

evaluation, context-evolution baselines keep updating after each task and Sentry after each verified recovery; only held-out tasks are scored. Retrieval details are in Appendix A.8.2.

5 Results

5.1 Main results

Sentry improves over both runtime intervention and context evolution baselines. Sentry improves on the strongest runtime-intervention baseline for each benchmark by 37% on average (up to 77% on WebShop; Table 1) and on ACE, the strongest context-evolution baseline, by 39% (Table 2). Adding Sentry to ACE gives the best result on both benchmarks, suggesting that task-level and failure-level knowledge are complementary. Secondary metrics are in Appendix A.5.1.

5.2 Intervention effectiveness and human validation

Sentry recovers diverse failure modes. Sentry resolves 81.7% of the 939 detected failures (Figure 3). Action-validity failures are the most reliably repaired (92.6%) because success is immediate and binary: the corrected action is either accepted or rejected by the environment. Progress (64.0%) and reasoning–grounding (70.2%) failures are harder because they require correcting the agent's behavior over a longer horizon, rather than fixing a single invalid action. Progress recovery is especially difficult on SWE-bench Lite (34.0%), where long repository-level tasks make it harder for the agent to get back on track once it gets stuck.

Soft repair improves local recovery and final task performance. To further evaluate Sentry's soft repair, we fork 200 soft-repair intervention checkpoints across the four benchmarks: from the same agent state, one branch receives Sentry's soft-repair intervention and the other continues unaided,

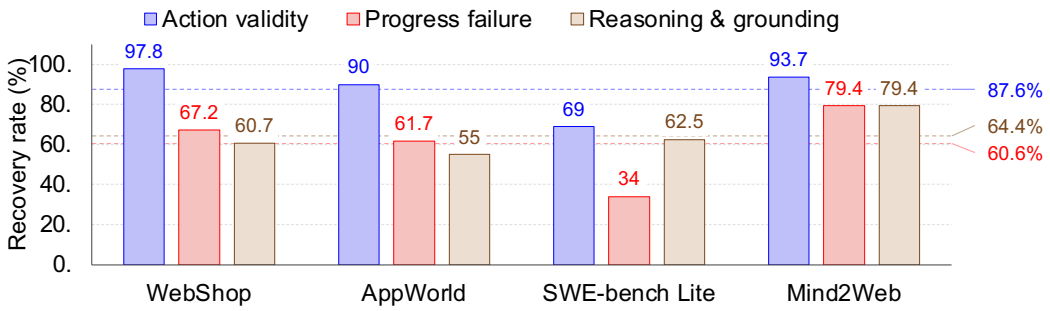


Figure 3: **Local recovery rates by failure category and benchmark.** Recovery rates measure the fraction of detected failures judged resolved within each category and benchmark. Action-validity recovery requires a corrected action accepted by the environment; progress and reasoning–grounding recovery is assessed by Sentry's verifier after (H=10) reasoning–action–observation cycles.

with no further interventions in either. Under the same recovery-verification criterion, local recovery increases from 33.5% without intervention to 60.5% with Sentry. The intervened branch achieves a higher final score in 74 of the 200 pairs, ties in 101, and performs worse in 25. This shows the benefit of Sentry's soft repair to both local recovery and final task performance. Full results are provided in Appendix A.6.1.

Sentry's intervention agrees with human judgements. Because the recovery rates above rely on Sentry's own detector and verifier, we validate them against human annotations on 200 trajectory windows, including triggered windows and task-matched non-triggered controls. Human agreement is 86.0% for intervention decisions (200 windows) and 81.8% for recovery judgments (99 soft-repair windows); 85.8% of the 113 issued interventions are judged locally helpful. Inter-annotator agreement is high (Cohen's $\kappa = 0.82$–0.89). Full annotation procedures and per-benchmark results are provided in Appendix A.7.

5.3 Ablations

All components contribute to Sentry's performance. We perform component ablations by removing one Sentry component at a time (Table 3); every ablation reduces performance on both benchmarks. Removing soft repair reduces performance by 41.9% on WebShop and 30.3% on AppWorld. Removing the intervention-need check yields the largest average relative drop across the two benchmarks (54.7%), highlighting the importance of selective intervention. The playbook ablations show another pattern: removing self-evolution, failure-specific retrieval, and recovery verification reduces performance by 31%, 26%, and 20% on average, respectively. Together, these results suggest that Sentry benefits largely from accumulating recovery experience, with further gains from verifying what is learned and retrieving it only for relevant failures. The variant definitions and additional ablation results are provided in Appendix A.6.2.

Recovery knowledge remains useful on held-out tasks. We initialize the playbook on 50 tasks and evaluate on 50 disjoint tasks (Table 4). With updates disabled, the frozen playbook improves WebShop reward from 0.251 to 0.364 and Mind2Web Replay completion from 0.533 to 0.651, compared with disabling retrieval. Continuing verified updates further increases these scores to 0.495 and 0.729. These results support reuse of recovery knowledge across disjoint tasks within the evaluated settings and the additional benefit of continued adaptation.

Persistent playbook exposure lowers mean performance. To examine exposure separately from memory evolution, we compare two variants using the same frozen playbook and runtime repair mechanisms. The exposure variant additionally provides the complete playbook as persistent background context to the task agent, while retaining failure-triggered retrieval and intervention. This lowers mean WebShop reward from 0.364 to 0.296, and mean Mind2Web Replay completion from 0.651 to 0.599. The absolute decreases are 0.068 and 0.052, respectively. This comparison supports

Table 3: Component ablations on WebShop and AppWorld. Each entry is the mean primary metric over the same five repeated runs as Table 1; the absolute drop is relative to Full Sentry. Additional ablation details are provided in Appendix A.6.2.

Variant	WebShop		AppWorld	
	Avg. reward	Drop	Test pass(%)	Drop
Full Sentry	**0.468**	–	**38.53**	–
w/o hard repair	0.297	0.171	31.20	7.33
w/o soft repair	0.272	0.196	26.85	11.68
w/o intervention-need check	0.136	0.332	23.70	14.83
w/o playbook failure-specific retrieval	0.316	0.152	31.05	7.48
w/o playbook self-evolution	0.295	0.173	28.90	9.63
w/o verification of recovery success	0.343	0.125	33.50	5.03

Table 4: Held-out evaluation of playbook reuse, persistent exposure, and continued adaptation (§4.3). The *No external playbook* variant retains Sentry's detector and repair routes but disables retrieval. The frozen variants and Full Sentry start from the same playbook initialized before evaluation, unlike the online-from-scratch protocol in Table 3. Both frozen variants disable updates and retain failure-triggered retrieval; they differ only in whether the complete playbook is additionally included in the task agent's persistent context. Full Sentry continues adding entries after verified soft-repair recoveries. Results are mean $\pm$ standard deviation over five runs.

Variant	WebShop avg. reward	Mind2Web Replay completion
Base agent	0.186 ± 0.014	0.421 ± 0.026
No external playbook	0.251 ± 0.017	0.533 ± 0.019
Frozen playbook	0.364 ± 0.019	0.651 ± 0.023
Frozen playbook + persistent exposure	0.296 ± 0.017	0.599 ± 0.018
Full Sentry	$\mathbf{0.495 \pm 0.017}$	$\mathbf{0.729 \pm 0.019}$

keeping the complete playbook outside the task agent's background context while providing relevant guidance during interventions. The detailed control is described in Appendix A.6.3.

Playbook retrieval works best when matched to the diagnosed failure. Given that the playbook contains reusable recovery knowledge, we next ablate how Sentry selects entries for soft repair. As retrieval becomes less specific to the current failure, performance consistently drops. Restricting retrieval only by the broad failure type reduces performance by approximately 11.3% (relative to Full Sentry, averaged over the two benchmarks); retrieving from the full playbook without filtering reduces it by approximately 18.8%; and randomly selecting playbook entries reduces it by approximately 28.9%. This shows that the playbook is most useful when Sentry retrieves experience that matches the failure it has diagnosed.

Performance is less sensitive to the number of retrieved entries: under the same calculation, $k = 3$ and $k = 10$ reduce performance by approximately 4.5% and 6.2%, respectively, relative to $k = 5$. We therefore use label-filtered retrieval with $k = 5$ as the default setting. Full results are in Appendix A.6.4.

Sentry is robust to context-window size and verification horizon. On WebShop, reward stays within 0.431–0.468 for $W \in \{3, 5, 7\}$ and 0.458–0.468 for $H \in \{5, 10, 20\}$, dropping only when the window is too short or too long (W=1: 0.352; W=15: 0.384) or verification comes too early (H=3: 0.432; Appendix A.6.5).

Table 5: **Token usage and latency.** Panel (a) reports input and output tokens in millions; the token ratio is each method's total divided by the base-agent total. Panel (b) reports the difference in per-task wall-clock time relative to the base agent, in seconds; negative values indicate shorter runtimes.

(a) Token usage by benchmark.

Method	WebShop	AppWorld	SWE-bench Lite	Mind2Web Replay	Total	Rel. cost
Base agent	6.19M	4.22M	1.06M	0.86M	12.32M	1.00×
Wink	6.24M	4.23M	1.07M	0.88M	12.42M	1.01×
AgentGuard	7.53M	5.39M	1.34M	1.01M	15.26M	1.24×
AgentFixer	8.20M	5.61M	1.41M	1.11M	16.33M	1.33×
Sentry	**9.30M**	**6.07M**	**1.73M**	**1.91M**	**19.01M**	**1.54×**
AgentForesight	15.32M	10.06M	2.48M	2.22M	30.08M	2.44×

(b) Per-task latency overhead relative to the base agent.

Method	WebShop delay	AppWorld delay	SWE-bench delay	Mind2Web Replay delay
Base agent	0.00	0.00	0.00	0.00
AgentGuard	-3.78	-26.06	-14.39	2.70
AgentForesight	22.98	-34.70	-5.35	-1.83
AgentFixer	-12.54	-57.54	-19.01	28.63
Wink	-1.70	-47.76	-15.24	-4.25
Sentry	**7.12**	**-45.61**	**-31.73**	**46.04**

5.4 Efficiency and robustness

Sentry's performance gains remain robust across models. With the larger GPT-OSS-120B as both the task agent and Sentry's component model, Sentry remains the strongest method, outperforming the strongest runtime-intervention baseline by 51% on average (95% on WebShop, 8% on Mind2Web Replay; Appendix A.5.2). The larger agent still loops and reasons without grounding, and Sentry's isolated failure management continues to improve performance.

Sentry adds moderate token usage but can reduce time on long-horizon tasks. Across the four benchmarks, Sentry's token usage is $1.54\times$ that of the base agent, versus $2.44\times$ for AgentForesight (Table 5a). Wall-clock time moves in both directions: per task, Sentry adds 7.1s on WebShop and 46.0s on Mind2Web Replay, but removes 31.7s on SWE-bench Lite and 45.6s on AppWorld (Table 5b). This suggests that on long tasks like SWE-bench, where looping can waste much of the budget, recovering stalled trajectories can save more time than Sentry spends on recovery.

6 Conclusion and limitations

Conclusion. Failure knowledge is conditional knowledge and should be conditionally exposed. Sentry puts this principle into practice: it keeps an evolving recovery playbook outside the agent's context, surfaces a few lessons only when it diagnoses a matching failure, and admits a lesson only after verifying the recovery. It outperforms runtime-intervention baselines on four benchmarks and context-evolution baselines on two, its lessons transfer to held-out tasks, and exposing the same playbook persistently lowers performance.

Limitations and future work. Our experiments use Qwen3.5-9B and GPT-OSS-120B; failure patterns may differ for other models and agent settings. Sentry uses the same model for the task agent, detector, and verifier; although its judgments largely agree with human annotators, decoupled detector and verifier models remain to be studied. Finally, the playbook may accumulate redundant or overly specific entries, motivating confidence-based admission, merging, and pruning.

AI USE STATEMENT

In this work, we used generative AI tools to help implement methods (LLMs were used to draft portions of the Sentry pipeline and evaluation harness) and to clean and reformat benchmark trajectory data for analysis. We did not use generative AI tools to develop the conceptual framework, propose or refine hypotheses, design the methodology or experiments, or interpret results; generating synthetic datasets, formulating or proving mathematical claims, translation, and qualitative/thematic analysis are not applicable to this work. Note that LLMs are also the object of study: the agents and the Sentry monitor are LLM-based systems, and their outputs are reported as experimental results rather than as author contributions. Additionally, we used generative AI tools to edit the paper for readability, to format references, and to assist with plotting scripts for figures. We have reviewed all AI-assisted work: LLM-generated code was checked and tested for correctness, and all edited prose was checked against our original drafts for accuracy. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Sentry aims to improve recovery in agent systems, but an intervention layer should still be used with care. A wrong diagnosis can constrain the agent in the wrong direction, add misleading recovery rules to the playbook, or shift the agent away from the original task objective. Since Sentry stores entries derived from past trajectories, any deployment on real user interactions should treat the playbook as sensitive memory and apply clear policies for filtering, access control, and deletion. Our experiments use established benchmarks and do not include private user data. Two members of the research team annotated the sampled benchmark trajectory windows. Annotation disagreements were resolved through discussion to reach consensus. We therefore view Sentry as a reliability aid for agent execution, not as a stand-alone safety guarantee.

REPRODUCIBILITY STATEMENT

The code we use, along with artifacts and experiment traces, will be released upon publication of the paper. We provide detailed descriptions of our experimental setup, including datasets, benchmarks, evaluation metrics, baselines, and hyperparameter choices. Additional details, such as prompts for large language models and implementation details, are included in the appendix. With this information, readers with reasonable computational resources should be able to reproduce our results.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r. ICLR 2026 Oral.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025.

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, and Qing Wang. From failed trajectories to reliable llm agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Sunny Dubey. Real-time detection and repair of LLM agent failures, 2026. URL https://arxiv.org/abs/2608.02464.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing, 2023. URL https://arxiv.org/abs/2305.11738.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Roham Koohestani. AgentGuard: Runtime verification of AI agents. *arXiv preprint arXiv:2509.23864*, 2025. doi: 10.48550/arXiv.2509.23864. URL https://arxiv.org/abs/2509.23864.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. Autoharness: Improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Hadar Mulian, Sergey Zeltyn, Ido Levy, Liane Galanti, Avi Yaeli, and Segev Shlomov. AgentFixer: From failure detection to fix recommendations in LLM agentic systems. *arXiv preprint arXiv:2603.29848*, 2026. doi: 10.48550/arXiv.2603.29848. URL https://arxiv.org/abs/2603.29848.

Rahul Nanda, Chandra Maddila, Smriti Jha, Euna Mehnaz Khan, Matteo Paltenghi, and Satish Chandra. Wink: Recovering from misbehaviors in coding agents. *arXiv preprint arXiv:2602.17037*, 2026. doi: 10.48550/arXiv.2602.17037. URL https://arxiv.org/abs/2602.17037. Also available on OpenReview.

OpenAI. Introducing gpt-oss. https://openai.com/index/introducing-gpt-oss/, 2025.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm. ICLR 2026 Poster.

Qwen Team. Qwen3.5-9B. https://huggingface.co/Qwen/Qwen3.5-9B, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Harsh Trivedi et al. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL https://arxiv.org/abs/2305.16291.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=NTAhi2JEEE.

Cailin Winston and René Just. A taxonomy of failures in tool-augmented LLMs. In *Proceedings of the 2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 125–135, 2025. doi: 10.1109/AST66626.2025.00019. URL https://doi.org/10.1109/AST66626.2025.00019.

Tianshi Xu, Huifeng Wen, and Meng Li. Adapting the interface, not the model: Runtime harness adaptation for deterministic llm agents. *arXiv preprint arXiv:2605.22166*, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Ye Yu, Xiaopeng Yuan, Haibo Jin, Heming Liu, Yaoning Yu, and Haohan Wang. Do self-evolving agents forget? capability degradation and preservation in lifelong llm agent adaptation. *arXiv preprint arXiv:2605.09315*, 2026. URL https://arxiv.org/abs/2605.09315.

Boxuan Zhang, Jianing Zhu, Zeru Shi, Dongfang Liu, and Ruixiang Tang. AgentForesight: Online auditing for early failure prediction in multi-agent systems. *arXiv preprint arXiv:2605.08715*, 2026a. doi: 10.48550/arXiv.2605.08715. URL https://arxiv.org/abs/2605.08715.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026c. URL https://openreview.net/forum?id=eC4ygDs02R. ICLR 2026 Poster.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners, 2023. URL https://arxiv.org/abs/2308.10144.

Kunlun Zhu et al. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

A Appendix

A.1 Additional Related Work

A.1.1 Context Evolution

As discussed in Section 2, context-evolution methods accumulate reusable experience across tasks and expose it to the task agent during future executions. Representative approaches include Reflexion, AWM, ReasoningBank, and ACE, which store experience as reflections, workflows, reasoning strategies, or evolving playbooks (Shinn et al., 2023; Wang et al., 2025; Ouyang et al., 2026; Zhang et al., 2026c). Related approaches such as ExpeL and Voyager similarly reuse prior experience through retrieved insights or executable skills (Zhao et al., 2023; Wang et al., 2023), while GEPA evolves prompts from execution feedback (Agrawal et al., 2026).

Sentry differs in the role and activation of the stored knowledge. Its persistent memory is restricted to failure-recovery experience and remains outside the task agent's persistent context. Relevant recovery lessons are introduced only after a matching runtime failure is detected. This design is complementary to task-level context evolution, as examined by our ACE+Sentry experiments.

A.1.2 Runtime Intervention

Runtime-intervention methods instead operate on the current execution, monitoring trajectories and providing corrective feedback when intervention criteria are met. As summarized in the main text, AgentGuard, AgentForesight, Wink, and AgentFixer use different combinations of runtime monitoring, failure prediction, validation, and corrective guidance (Koohestani, 2025; Zhang et al., 2026a; Nanda et al., 2026; Mulian et al., 2026). Earlier feedback-driven methods such as Self-Refine and CRITIC establish related mechanisms for improving a current attempt through critique or external feedback (Madaan et al., 2023; Gou et al., 2023). Dubey (2026) additionally studies lightweight monitoring, deterministic checks, and rollback-and-rerun repair.

Sentry builds on the runtime-intervention setting but additionally retains verified recovery experience across tasks. Rather than generating each intervention only from the current trajectory, Sentry can retrieve recovery principles learned from earlier successful repairs.

A.1.3 Harness Adaptation and Failure Diagnosis

A complementary line of work improves reliability by modifying the harness that mediates observations, tool use, and execution. Life-Harness, AutoHarness, Self-Harness, and HarnessFix adapt or synthesize harness components from observed failures (Xu et al., 2026; Lou et al., 2026; Zhang et al., 2026b; Chen et al., 2026). Sentry does not modify its monitoring, routing, or retrieval procedures during execution; its evolving state is the external recovery knowledge used by those procedures.

Failure-analysis work instead characterizes where agent executions break down. MAST studies failure modes in multi-agent systems (Cemri et al., 2025), while AgentErrorTaxonomy, AgentErrorBench, and AgentDebug analyze failures across planning, reasoning, memory, action, and system-level execution (Zhu et al., 2025). Sentry uses a narrower, operational failure categorization: failure types determine whether to intervene, which repair route to use, and which recovery lessons to retrieve.

A.2 Detailed Motivating Examples

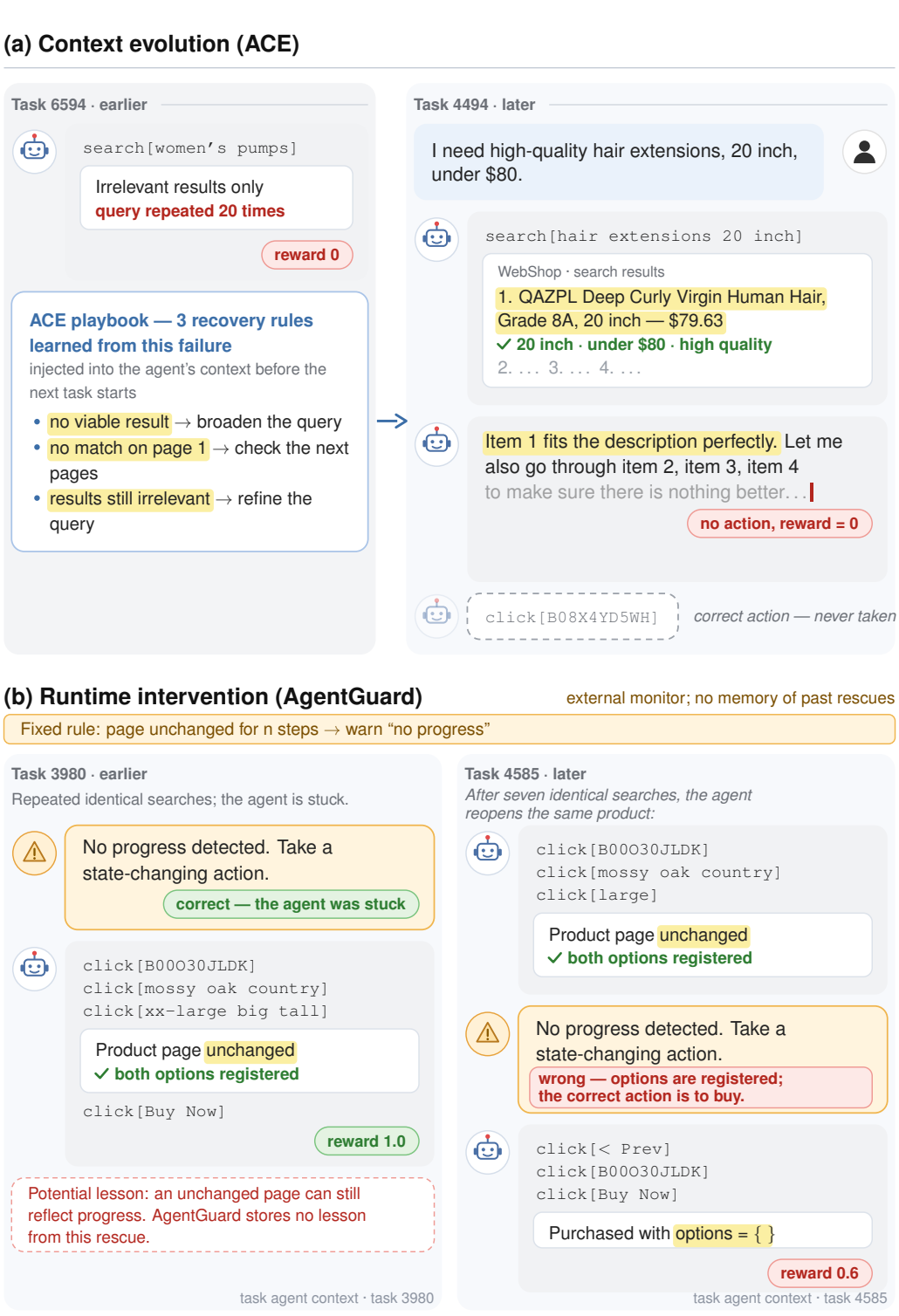


Figure 4: **Detailed examples underlying Figure 1.** Trajectories from Qwen3.5-9B on WebShop. (a) ACE learns recovery rules from task 6594 and retains them in the agent's context for task 4494. Despite identifying a matching item, the agent continues comparing alternatives and emits no action. (b) AgentGuard's fixed warning helps the agent recover from repeated searches in task 3980, but no recovery lesson is stored. In task 4585, the same warning fires after valid option selections leave the page unchanged. The agent then navigates away and purchases the item without those options.

As shown in Figure 4, we expand the examples in Figure 1, showing the recovery rules, agent actions, and task outcomes in greater detail. Both examples use Qwen3.5-9B on WebShop.

A.3 Experimental Setup Details

This section provides additional details for the experimental setup in Section 4.1 and Section 4.3. We first describe benchmark-specific details and metrics, then the baseline implementations, and finally the evaluation protocols used throughout the experiments.

A.3.1 Benchmarks and Metrics

Benchmark details. We evaluate on WebShop, AppWorld, SWE-bench Lite, and Mind2Web Replay, using the benchmark interfaces and primary metrics described in Section 4.1. The main runtime-intervention comparison uses 100 tasks each for WebShop, AppWorld, and Mind2Web Replay, and 50 tasks for SWE-bench Lite.

Before sampling, we exclude tasks that cannot be executed in our evaluation setup because of environment or dependency issues, as well as tasks with missing required annotations. Tasks are randomly sampled from the remaining eligible pool. The selected task sets remain fixed across compared methods and across repeated runs.

WebShop reports average reward as its primary metric. We additionally report the number of tasks achieving full success (reward $= 1$) and the number of tasks in which the agent completes a purchase (*bought*). AppWorld reports the task test-pass rate: the percentage of evaluation checks passed per task, averaged across tasks. For SWE-bench Lite, we evaluate generated patches using the official SWE-bench evaluation harness with its Modal backend and report the issue-resolution accuracy. Mind2Web Replay reports mean normalized trajectory completion, defined below.

Mind2Web Replay protocol. *Purpose and relation to the original benchmark.* Mind2Web provides task instructions, annotated action sequences, and recorded webpage states from real web tasks (Deng et al., 2023). In the original evaluation, each reference step is evaluated independently: the model receives the recorded state and preceding reference history, then predicts the next action. This measures next-action prediction, but does not evaluate how an agent recovers after taking an incorrect action.

Because Sentry is designed for runtime failure detection and recovery, we adapt Mind2Web to allow such recovery while preserving the original tasks, recorded states, and annotated target actions. In Mind2Web Replay, a rejected action leaves the agent at the same reference state and returns feedback, so the agent can revise its next attempt and, when applicable, receive runtime intervention. The environment advances only after an accepted action.

This preserves the benchmark's core task of selecting the correct action at each recorded state, while additionally evaluating recovery from execution-time mistakes. Because Replay allows retries and trajectory-level progress, its scores are not directly comparable to the original Mind2Web step-wise metrics.

Data and task construction. We use the `test_task` split of `osunlp/Multimodal-Mind2Web`, corresponding to the Cross-Task setting.[1] We reconstruct each task by grouping action-level records by `annotation_id` and ordering them by `target_action_index`. We exclude tasks containing a reference step with no positive candidate element, since such steps cannot be completed through our candidate-based interface. From the remaining tasks, we sample 100 for the main evaluation; held-out experiments use 50 initialization tasks and 50 disjoint evaluation tasks.

Interaction and action evaluation. At each reference step, the agent observes the task goal and a candidate set containing all annotated positive elements and up to 30 negative elements. Candidates are presented as textual descriptions and assigned labels such as `A`, `B`, and `AA`. The agent can issue `click`, `type`, or `select` actions over these labels.

An action is accepted only if it selects an annotated positive candidate, uses the correct operation, and, for `TYPE` or `SELECT`, matches the annotated value. Accepted actions advance to the next recorded

[1] https://huggingface.co/datasets/osunlp/Multimodal-Mind2Web

reference state. Rejected actions leave the agent at the current state and return annotation-derived feedback, allowing another attempt. The same interface and transition rules are used for all compared methods.

Scoring. For task i, let N_i be the number of reference steps and C_i the number completed in order before termination. We score each task as

$$r_i = \frac{C_i}{N_i}, \tag{1}$$

and report mean normalized trajectory completion,

$$\overline{r} = \frac{1}{M} \sum_{i=1}^{M} \frac{C_i}{N_i}. \tag{2}$$

We also report the number of tasks whose full reference trajectory is completed. Partial progress is retained when execution terminates before the final step.

Scope. Mind2Web Replay follows the recorded reference trajectory rather than executing arbitrary actions on a live website. It therefore evaluates recovery under a controlled, annotation-conditioned interaction setting. The benchmark preserves the original task instructions, recorded states, and annotated target actions, but adds retry-based interaction after incorrect actions. Its scores should therefore not be compared directly with the original Mind2Web step-wise metrics.

Additional metric definitions. In addition to benchmark-specific task metrics, we report several intervention-level diagnostics.

Number of interventions (# Interv.). The total number of interventions issued by a method across evaluated tasks.

Guided tasks. The number of tasks receiving at least one intervention.

Guidance lift. For intervention methods, guidance lift reports the average performance improvement relative to the base agent on the tasks receiving intervention. Because methods may intervene on different subsets of tasks, guidance lift is used as a descriptive intervention-level diagnostic rather than as the primary metric for comparing methods.

For SWE-bench Lite, where the task-level outcome is binary, we additionally report the number of guided tasks that are recovered after intervention.

A.3.2 Baseline Implementations

Shared model configuration. We evaluate Sentry and the runtime-intervention baselines under a shared-backbone configuration. In the main experiments, the task agent and all LLM-based auxiliary components use Qwen3.5-9B, including detection, auditing, guidance generation, and memory updates where applicable. Across the compared methods, task-agent outputs are limited to 384 generated tokens per call, including any action regeneration or continuation following an intervention.

We use the same language-model backbone for the task agent and all LLM-based auxiliary components so that differences in performance reflect the intervention methods rather than differences in model capability. Each baseline otherwise retains its method-specific monitoring, intervention, state, and update procedures. Unless stated otherwise, tasks are evaluated independently, with no access to future task trajectories or outcomes.

Runtime intervention baselines. *AgentGuard.* Our AgentGuard implementation follows (Koohestani, 2025). The monitor processes each executed environment transition and maintains statistics over observed states, actions, transition outcomes, invalid actions, no-progress transitions, and repeated no-progress actions. It issues a warning when the invalid-transition rate exceeds 0.25, the no-progress rate exceeds 0.75, or the same no-progress action occurs at least three times consecutively. A new monitor is initialized for each task. A valid progress-making action resets the repeated no-progress-action streak, but does not clear the accumulated transition statistics.

AgentFixer. We use the implementation following arXiv:2603.29848v1 (Mulian et al., 2026).

AgentFixer validates each proposed action using the applicable components of its validator suite, including unusual-token detection, repetition detection, output-schema and output-format validation, information-consistency and action-grounding validation, and reasoning–action alignment. Findings are assigned *minor*, *moderate*, or *critical* severity.

When the maximum detected severity is moderate or critical, diagnostic remediation feedback is returned to the task agent and the action is regenerated. The validator is stateless and checks each candidate independently. At most one regeneration is allowed per environment step, with the retry count resetting at the next step.

Wink. We use the LLM-based trajectory-monitoring and course-correction framework of Wink (Nanda et al., 2026), following arXiv:2602.17037v2.

Under the shared-backbone configuration, Wink's detector uses the same Qwen3.5-9B model as the task agent. The detector is invoked every five agent actions and examines the eight most recent trajectory steps for failures such as specification drift, reasoning problems, and tool-call errors. Monitoring operates throughout execution on all four benchmarks, including intermediate execution steps on SWE-bench Lite.

When a failure is detected, the diagnosis and supporting trajectory evidence are converted into category-specific corrective guidance. The guidance is appended to the task agent's conversation within `<system-reminder>` tags. Each task begins with a fresh trajectory history. Issuing a reminder does not clear the detector's recent-trajectory window.

AgentForesight. We use the official AgentForesight inference implementation (Zhang et al., 2026a) at commit `af54908`, following its incremental-prefix prompting and structured-verdict protocol. Before an action is executed, the auditor receives the executed trajectory prefix together with the proposed action. A valid non-`SAFE` verdict blocks the candidate, and its reported failure reason is returned to the task agent through a domain-specific correction instruction. The regenerated candidate is then audited again.

Auditor calls are stateless, with all context supplied explicitly. Rejected candidates are excluded from the executed trajectory; only executed actions and their resulting observations are included in subsequent trajectory prefixes.

Context evolution baselines. We use the official implementations of Reflexion, ACE, AWM, and ReasoningBank at the commits listed below, with their method-specific memory construction, retrieval, and update procedures. All LLM-based components use the shared task model.

Reflexion. We use the official Reflexion implementation (Shinn et al., 2023) at commit `218cf0e`. Reflexion maintains episodic natural-language reflections derived from task execution. Its reflection procedure uses the task instruction, execution trajectory, and feedback to identify failures and formulate lessons for subsequent tasks. Retrieved reflections are supplied to the task agent before execution, and new reflections are incorporated after completed tasks.

ACE. We use the official ACE implementation (Zhang et al., 2026c) at commit `82709de`. ACE maintains an evolving playbook of reusable task-level advice. Execution feedback is used to revise and extend the playbook through its generation, reflection, and curation procedures. Before each task, retrieved playbook guidance is supplied through the repository's context-construction procedure. The playbook continues to evolve as tasks are completed.

AWM. We use the official AWM implementation (Wang et al., 2025) at commit `8c0ff8c`. AWM extracts reusable procedural workflows from task trajectories, abstracting task-specific details into guidance that can be applied to later tasks. Selected workflows are provided to the task agent before execution. The workflow memory is updated from completed tasks using the official workflow-induction procedure.

ReasoningBank. We use the official ReasoningBank implementation (Ouyang et al., 2026) at commit `ed80611`. ReasoningBank distills task trajectories into reusable reasoning insights, learning

from both successful and failed experiences. Relevant entries are supplied to the task agent before each task. Experience from completed tasks is subsequently incorporated through the official memory-construction and update procedures.

ACE+Sentry. ACE+Sentry combines task-level context evolution with Sentry's external runtime failure management. The two methods maintain separate memories and retain their respective activation and update rules.

ACE maintains its task-level playbook and supplies retrieved entries to the task agent's initial context before execution. Sentry independently maintains its recovery playbook outside the task agent's persistent context and retrieves from it only when a soft-repair intervention is triggered.

During held-out evaluation, ACE updates its playbook after each completed task according to its standard update procedure. Sentry adds new recovery entries only after verified soft-repair recoveries. The two playbooks are not merged.

A.4 Implementation details

Serving. All model calls, including those made by the task agent, Sentry, and LLM-based baselines, are served through the Together API.

Task selection and repeated runs. We use the official benchmark releases and evaluation splits available at the time of our experiments. Before sampling, we exclude tasks that cannot be executed in our evaluation setup because of environment or dependency issues, as well as tasks with missing required annotations. Tasks are randomly sampled from the remaining eligible pool. The selected task sets remain fixed across five repeated runs, while the task execution order is randomized for each run. Under the held-out protocol, the initialization and evaluation sets are fixed before the repeated runs; execution order is randomized within each set, and initialization completes before held-out evaluation begins. All updates use only information available at the time.

Sentry configuration. In the default configuration, the task agent and all Sentry components use the same Qwen3.5-9B backbone. The task agent uses a temperature of 0.7 and a maximum of 384 generated tokens per call, including action regeneration during hard repair and continuation after soft-repair guidance. Sentry's auxiliary calls, including failure detection, recovery verification, and playbook updating, use a temperature of 0 and a maximum generation length of 4096 tokens. Each component produces structured outputs with fixed fields following the schemas in Appendix A.8.3. The detector sees the task objective, action schema, and recent trajectory window; the verifier sees only the triggering failure and the subsequent H-step recovery trajectory. Neither observes benchmark rewards or final task success.

A.4.1 Evaluation Protocols

We use the two evaluation protocols defined in Section 4.3: an *online-from-scratch protocol* for the main runtime-intervention comparison and component ablations, and a *held-out protocol* for context-evolution comparisons and experiments involving pre-initialized recovery memory. This section provides additional implementation details.

Task selection and repeated runs. For each experiment, the sampled task set is fixed across all compared methods and across the five repeated runs. All methods within a comparison use the same task set, environment interface, task-agent model, and execution limits. Tasks that cannot be executed in our setup because of environment or dependency failures, or that lack required annotations, are removed before sampling. All methods use task-agent temperature 0.7, with fresh generation samples and randomized task order across runs on a fixed task set.

Online-from-scratch protocol. As described in Section 4.3, Sentry starts each run with an empty recovery playbook. Playbook entries available at any point therefore come only from verified soft-repair recoveries encountered earlier in that run. No recovery knowledge from future tasks or outcomes is available.

Table 6: Full WebShop results using Qwen3.5-9B.

Method	Avg. reward	Full succ.	Bought	# Interv.	# Guided tasks	Guid. lift
Base agent	0.170 ± 0.016	6	23	0	0	–
AgentGuard	0.258 ± 0.019	7	37	533	65	0.134
AgentForesight	0.168 ± 0.011	4	25	15	14	-0.018
AgentFixer	0.171 ± 0.017	5	24	236	46	0.002
Wink	0.264 ± 0.015	10	33	137	60	0.156
Sentry	$\mathbf{0.468 \pm 0.017}$	**14**	**79**	227	**93**	**0.320**

Table 7: Full AppWorld results using Qwen3.5-9B. Guidance lift is in percentage points of test pass rate.

Method	Test pass (%)	# Interv.	# Guided tasks	Guid. lift
Base agent	25.59 ± 0.88	0	0	–
AgentGuard	25.09 ± 0.91	630	54	-0.93
AgentForesight	24.17 ± 0.91	25	12	-11.83
AgentFixer	26.69 ± 0.80	14	8	13.75
Wink	27.56 ± 0.98	24	14	14.07
Sentry	$\mathbf{38.53 \pm 0.87}$	187	**66**	**19.61**

Component-ablation variants use the same task sequence and initialization as Full Sentry. Unless an ablation explicitly modifies memory construction, retrieval, or verification, the remaining Sentry components follow the Full Sentry procedure.

Held-out protocol. The held-out protocol uses 50 initialization tasks followed by 50 disjoint evaluation tasks, with only the evaluation tasks contributing to reported scores (Section 4.3). During evaluation, methods continue updating their memories according to their own update procedures unless the experiment explicitly freezes memory.

For Sentry, updates occur only after verified soft-repair recoveries. For context-evolution baselines, updates occur according to each method's task-level memory-update procedure after completed tasks. In all cases, updates use only information observed up to the current point in the task sequence.

For the frozen-playbook exposure control, both variants begin evaluation from the same playbook produced during initialization and disable all further playbook updates. They use the same task order, detector, repair procedures, retrieval rule, and recovery verifier. The only difference is whether the complete frozen playbook is additionally placed in the task agent's persistent context. Failure-triggered retrieval remains enabled in both conditions.

A.5 Additional Main Results

This section provides the complete results underlying the main comparisons in Section 5.1. We first report benchmark-specific secondary metrics and intervention diagnostics, followed by results with GPT-OSS-120B and the efficiency analysis.

A.5.1 Full Benchmark Results

Tables 6–9 report the complete results for the runtime-intervention comparison in Table 1. In addition to each benchmark's primary metric, we report benchmark-specific secondary metrics and the intervention-level diagnostics defined in Appendix A.3.

Table 8: Full SWE-bench Lite results using Qwen3.5-9B.

Method	Acc.	# Interv.	# Guided tasks	# Recovered
Base agent	0.20 ± 0.01	0	0	–
AgentGuard	0.22 ± 0.02	6	3	1
AgentForesight	0.24 ± 0.01	22	15	3
AgentFixer	0.22 ± 0.01	75	38	4
Wink	0.18 ± 0.02	4	3	3
Sentry	$\mathbf{0.30 \pm 0.01}$	90	9	**5**

Table 9: Results on Mind2Web Replay using Qwen3.5-9B. Mean replay completion is the average fraction of reference steps completed in order. Fully completed tasks counts reference trajectories completed in their entirety. The protocol is specified in Appendix A.3.1.

Method	Mean replay completion	Fully completed tasks	# Interv.	# Guided tasks	Guid. lift
Base agent	0.420 ± 0.030	32	0	0	–
AgentGuard	0.450 ± 0.019	37	448	94	0.032
AgentForesight	0.430 ± 0.016	34	470	93	0.011
AgentFixer	0.687 ± 0.027	50	517	70	0.382
Wink	0.370 ± 0.020	32	7	6	-0.840
Sentry	$\mathbf{0.736 \pm 0.019}$	**55**	435	77	**0.410**

A.5.2 Results with GPT-OSS-120B

To examine whether the main results depend on the default Qwen3.5-9B backbone, we repeat the runtime-intervention comparison on WebShop and Mind2Web Replay using GPT-OSS-120B for both the task agent and the LLM-based intervention components. The results are reported in Table 10.

Sentry obtains the highest measured primary task metric on both benchmarks in this setting. These results provide evidence that the gains observed with Qwen3.5-9B are not specific to that backbone, although the GPT-OSS-120B evaluation covers only two of the four benchmarks.

A.6 Additional Intervention Analysis and Ablations

This section provides the full intervention-level analyses underlying the results in Section 5.2. In the main text, we show that Sentry recovers diverse failure modes and that soft repair improves both local recovery and final task performance. Here, we provide the complete paired soft-repair results and benchmark-level failure-recovery breakdown.

A.6.1 Paired Soft-Repair Evaluation

As described in Section 5.2, we fork 200 soft-repair checkpoints across the four benchmarks. At each checkpoint, both branches begin from the same agent state and trajectory history: one receives Sentry's soft-repair intervention and the other continues without intervention. No further interventions are allowed in either branch.

Local recovery is evaluated after the same $H = 10$ horizon used by Sentry's recovery verifier, and both branches are then continued to termination for the benchmark-specific final task metric.

Table 11 reports the complete results behind the main-text comparison. Because both branches share the same pre-intervention state, this experiment directly compares continuations with and without soft repair.

Table 10: Results using GPT-OSS-120B on WebShop and Mind2Web Replay. WebShop reports average reward and full-success/purchase counts; Mind2Web Replay reports mean normalized trajectory completion and the number of fully completed tasks.

Method	WebShop reward	WebShop succ./bought	WebShop guid. lift	Mind2Web Replay completion	Mind2Web Replay completed tasks	Mind2Web Replay guid. lift
Base agent	0.192	7/28	–	0.550	43	–
AgentGuard	0.246	8/35	0.092	0.740	68	0.194
AgentForesight	0.091	4/12	-0.186	0.582	52	0.033
AgentFixer	0.273	5/44	0.112	0.753	62	**0.328**
Wink	0.197	7/33	0.008	0.580	47	0.111
Sentry	**0.531**	**12/92**	**0.357**	**0.815**	**75**	0.294

Table 11: Paired counterfactual evaluation at 200 soft-repair checkpoints. Local recovery is judged after $H = 10$ steps. The final row counts pairs in which each branch obtains a higher final score under the task-level metric defined in Section 4.1; the remaining 101 pairs are tied.

	No intervention	**Sentry**
Local recovery (of 200)	67 (33.5%)	**121 (60.5%)**
Recovered only in this branch	12	**66**
Higher final task score	25	**74**

A.6.2 Component Ablations

We provide the full component-ablation results on WebShop and AppWorld, expanding the compact comparison reported in Table 3 in the main text. Tables 12 and 13 report the complete benchmark-specific results, including secondary metrics and intervention-level diagnostics.

All variants follow the online-from-scratch protocol in Appendix A.4.1 and use the same task sets and model configuration as Full Sentry. Each variant removes or modifies one component while leaving the remaining procedure unchanged unless the removed component directly affects downstream operations.

Without hard repair. We disable Sentry's action-correction procedure for schema-invalid actions. When an invalid action is detected, Sentry does not request a corrected replacement action. The soft-repair route, including failure detection, playbook retrieval, recovery verification, and playbook updates, remains enabled.

Without soft repair. We disable corrective guidance for progress and reasoning–grounding failures. Action-validity checking and hard repair remain enabled. Because Full Sentry adds playbook entries only from verified soft-repair episodes, this variant also removes the source of new recovery entries.

Without intervention-need check. We remove the binary gate that determines whether soft-repair guidance should be issued. For valid actions, guidance is generated at every scheduled monitoring point rather than only after the detector identifies a failure. The remaining repair, verification, and playbook-update procedures are unchanged.

Without failure-specific retrieval. We retain the evolving playbook but remove conditioning on the diagnosed failure type and fine-grained retrieval labels. Entries are instead selected without using the current failure diagnosis. Failure detection, soft repair, recovery verification, and playbook updates remain enabled.

Without playbook self-evolution. We disable all playbook updates. Under the online-from-scratch protocol, the playbook therefore remains empty throughout evaluation. Failure detection and both

Table 12: Component-ablation results on WebShop. Each variant removes one component from Sentry. Downward arrows indicate drops relative to Full Sentry for the primary task-level metric. Guidance lift is computed over guided tasks, following the definition in Appendix A.3.1.

Variant	Avg. reward	Full succ.	Bought	# Interv.	# Guided tasks	Guid. lift
Full Sentry	**0.468**	**14**	**79**	227	93	**0.320**
w/o hard repair	0.297 (↓ 0.171)	5	51	59	53	0.240
w/o soft repair	0.272 (↓ 0.196)	9	63	159	75	0.136
w/o playbook failure-specific retrieval	0.316 (↓ 0.152)	6	45	102	93	0.156
w/o playbook self-evolution	0.295 (↓ 0.173)	4	38	198	89	0.140
w/o intervention-need check	0.136 (↓ 0.332)	1	19	347	99	−0.035
w/o verification of recovery success	0.343 (↓ 0.125)	8	55	214	91	0.190

Table 13: Component-ablation results on AppWorld. Each variant removes one component from Sentry; arrows give the drop in test pass rate relative to Full Sentry. Guidance lift is in percentage points and follows the definition in Appendix A.3.1.

Variant	Test pass (%)	# Interv.	# Guided tasks	Guid. lift
Full Sentry	**38.53**	187	66	**19.61**
w/o hard repair	31.20 (↓ 7.33)	67	44	12.75
w/o soft repair	26.85 (↓ 11.68)	120	54	2.33
w/o playbook failure-specific retrieval	31.05 (↓ 7.48)	118	66	8.27
w/o playbook self-evolution	28.90 (↓ 9.63)	166	62	5.34
w/o intervention-need check	23.70 (↓ 14.83)	310	92	−2.05
w/o verification of recovery success	33.50 (↓ 5.03)	178	64	12.36

repair routes remain enabled, but soft repair relies only on the current diagnosis and general repair instructions.

Without recovery verification. We remove the resolved/unresolved recovery judgment as a prerequisite for playbook admission. Soft-repair episodes are added to the playbook after the same observation horizon without requiring a verified recovery. The repair and retrieval procedures otherwise remain unchanged.

Across both benchmarks, every component ablation reduces the primary task metric relative to Full Sentry. These results are consistent with complementary contributions from selective intervention, soft and hard repair, recovery-memory evolution, failure-conditioned retrieval, and verification before memory admission.

Full Sentry achieves a higher primary task metric than every ablated variant on both WebShop and AppWorld. The detailed results in Tables 12 and 13 are consistent with the main-text analysis: removing selective intervention, soft or hard repair, playbook evolution, failure-conditioned retrieval, or recovery verification reduces performance.

A.6.3 Held-Out Playbook Controls

The held-out playbook results are summarized in Section 5.3. Here, we clarify the construction of the variants used in that comparison.

All variants use the same 50 initialization tasks and 50 disjoint held-out evaluation tasks. They differ only in how the initialized playbook is used during evaluation.

No external playbook retains Sentry's failure detection and repair mechanisms but disables playbook retrieval. *Frozen playbook* starts evaluation from the playbook produced during initialization, allows failure-triggered retrieval from that playbook, and disables further playbook updates. *Frozen playbook*

Table 14: Playbook retrieval ablations on held-out WebShop and Mind2Web Replay tasks. Only the soft-repair retrieval rule is varied; all other Sentry components are fixed. We report mean ± standard deviation over five repeated runs. Label-filtered retrieval with $k = 5$ is the Full Sentry configuration and uses the same runs as Table 2 (0.495 and 0.729 at three decimals); relative decreases in the text are computed against those means.

Retrieval variant	WebShop avg. reward	Mind2Web Replay completion
Random retrieval	0.33 ± 0.03	0.55 ± 0.02
Unfiltered retrieval	0.39 ± 0.01	0.61 ± 0.02
Failure-type-only retrieval	0.43 ± 0.01	0.66 ± 0.01
Label-filtered, $k = 3$	0.47 ± 0.01	0.70 ± 0.01
Label-filtered, $k = 5$	$\mathbf{0.50 \pm 0.02}$	$\mathbf{0.73 \pm 0.02}$
Label-filtered, $k = 10$	0.46 ± 0.02	0.69 ± 0.02

+ *persistent exposure* uses the same frozen playbook and retrieval procedure, while additionally placing the complete playbook in the task agent's background context throughout execution. *Full Sentry* starts from the same initialized playbook and continues adding entries after verified soft-repair recoveries during held-out evaluation.

For the frozen-playbook comparison, the two conditions use the same task-agent model, task order, environment, failure detector, repair procedures, retrieval rule, and recovery verifier. The playbook remains frozen throughout evaluation, and the only intentional difference is that the persistent-exposure variant additionally places the complete playbook in the task agent's background context.

These variants jointly examine reuse of previously learned recovery knowledge, continued playbook evolution, and persistent versus failure-triggered exposure of the playbook.

A.6.4 Retrieval Ablations

Table 14 presents the full results summarized in Section 5.3. We clarify the retrieval variants used in that comparison.

All variants use the same held-out evaluation setup and differ only in how playbook entries are selected for soft repair. *Random retrieval* samples entries without conditioning on the current failure. *Unfiltered retrieval* selects from the full playbook without using the diagnosed failure type or fine-grained labels. *Failure-type-only retrieval* restricts selection to entries with the same broad failure type. *Label-filtered retrieval* additionally requires overlap with the fine-grained labels assigned to the current diagnosis.

For label-filtered retrieval, k denotes the maximum number of playbook entries included in a soft-repair prompt. We evaluate $k \in \{3, 5, 10\}$; $k = 5$ is the default Full Sentry setting. All other Sentry components are held fixed across these variants.

A.6.5 Hyperparameter Sensitivity

The hyperparameter sensitivity results are summarized in Section 5.3. Here, we provide the full results and clarify the two parameters varied in these experiments.

Context-window size W. The detector observes the most recent W reasoning–action–observation cycles when deciding whether a failure has occurred. We evaluate $W \in \{1, 3, 5, 7, 10, 15\}$ on WebShop, while keeping all other Sentry components and settings fixed. The full results, including intervention and guidance statistics, are reported in Table 15. The default configuration uses $W = 5$.

Recovery-verification horizon H. For soft repair, the recovery verifier observes H subsequent reasoning–action–observation cycles before judging whether the triggering failure has been resolved. We evaluate $H \in \{3, 5, 10, 20\}$ on WebShop, with all other settings fixed. The full results include

Table 15: Effect of Sentry context-window size on WebShop performance.

Context window	Avg. reward	Full succ.	Bought	# Interv.	# Guided tasks	Guid. lift
1	0.352	8	60	165	78	0.233
3	0.431	12	73	205	88	0.296
5	**0.468**	**14**	**79**	**227**	**93**	**0.320**
7	0.456	13	78	238	94	0.304
10	0.421	12	72	255	95	0.264
15	0.384	10	66	275	97	0.220

Table 16: Sensitivity to the soft-repair verification horizon H on WebShop. Performance and recovery-judge accuracy are stable across moderate verification horizons, while verifying too early reduces both end-to-end task performance and agreement with human recovery judgments.

H	Avg. reward ↑	Guidance lift ↑	Recovery-judge acc. (%) ↑
3	0.432	0.283	76.0
5	0.458	0.307	82.0
10	**0.468**	**0.320**	**84.0**
20	0.461	0.312	83.0

both task performance and recovery-verifier accuracy, reported in Table 16. The default configuration uses $H = 10$.

A.7 Human Evaluation

This appendix gives the full protocol and results for the human audit summarized in Section 5.2. The audit checks three of Sentry's runtime judgments against human assessment: the detector's decision to intervene, the verifier's decision that the triggering failure has been resolved, and the local helpfulness of the intervention text.

Sampling. We sample 200 trajectory windows from completed Sentry runs, 50 from each benchmark. Each window contains the recent trajectory seen by the detector and, when Sentry intervened, the intervention text and the subsequent H-step recovery trajectory. The sample contains 113 windows in which Sentry issued an intervention (99 soft repairs and 14 hard repairs, covering Progress, Reasoning & Grounding, and Action Validity failures) and 87 control windows in which it did not. Control windows are matched to the triggered windows by task: they are drawn from the same tasks and runs, at steps where the detector did not fire, so that triggered and control windows share the same task distribution. Including controls lets intervention accuracy reflect both false alarms and missed failures rather than false alarms alone.

Annotation procedure. Each window is independently labeled by two members of the research team using the rubric in Table 17. The two annotators then discuss any disagreements to reach consensus, and the resulting consensus labels are used for all Sentry-vs-human comparisons. Sentry's trigger decision is withheld during the first annotation stage, and its verifier verdict is withheld throughout annotation. Each window is presented in two stages. In the first stage, annotators see only the pre-intervention trajectory and assign the *intervention needed* label, without knowing whether Sentry intervened at that point; triggered and control windows are indistinguishable at this stage. In the second stage, for triggered windows only, the intervention text and the subsequent H-step continuation are revealed, and annotators assign the *recovered* label from the continuation (has the triggering failure pattern stopped, with the agent resuming valid, task-relevant progress?) and the *helpful* label from the intervention text and its local failure context (does the guidance address the observed failure and give task-relevant advice, regardless of whether the agent ultimately follows

Table 17: Human annotation rubric for auditing Sentry trajectory windows.

Label	Criterion
Intervention needed	The agent shows an invalid action, repeated no-progress behavior, poor grounding, or another failure that requires intervention.
Recovered	After Sentry's intervention, the agent stops the triggering failure pattern and resumes valid, task-relevant progress.
Helpful	The intervention text addresses the observed local failure and gives task-relevant guidance without unsupported assumptions or objective drift.

Table 18: Human validation of Sentry's intervention decisions, recovery judgments, and intervention helpfulness. N is the number of audited windows per benchmark. Intervention accuracy is computed over all windows, recovery-judge accuracy over soft-repair windows (99 in total), and helpful rate over issued interventions (113 in total). All values except N are percentages.

Benchmark	N	Intervention acc.	Recovery-judge acc.	Helpful rate
WebShop	50	86.0	84.0	86.2
AppWorld	50	86.0	81.0	85.2
SWE-bench Lite	50	82.0	76.9	82.1
Mind2Web Replay	50	90.0	85.2	89.7
Overall	200	86.0	81.8	85.8

it?). Annotators assign the *recovered* label based on the continuation without access to the verifier's resolved/unresolved verdict.

Metrics. We compare Sentry's decisions with the consensus labels using three metrics, each over a different denominator. *Intervention accuracy* is computed over all 200 windows: the fraction in which Sentry's trigger/no-trigger decision matches the intervention-needed label. *Recovery-judge accuracy* is computed over the 99 soft-repair windows: the fraction in which the verifier's resolved/unresolved output matches the recovered label. The 14 hard-repair windows are excluded from this metric because hard repair is verified by execution—the corrected action is either accepted by the environment or not—rather than by the LLM verifier, so there is no verifier judgment to audit. *Helpful rate* is computed over all 113 issued interventions: the fraction judged locally helpful.

Results. Table 18 reports results per benchmark. Overall, Sentry reaches 86.0% intervention accuracy, 81.8% recovery-judge accuracy, and an 85.8% helpful rate. Agreement is highest on Mind2Web Replay and lowest on SWE-bench Lite, and recovery-judge accuracy is the lowest of the three metrics on every benchmark, consistent with recovery being the hardest judgment: it requires assessing a multi-step continuation rather than a single state or a single message. These results indicate that Sentry usually intervenes on failures that humans also identify, that its recovery verdicts generally agree with human assessment, and that the guidance it issues is mostly judged useful.

Inter-annotator agreement. Table 19 reports raw percent agreement and Cohen's κ between the two annotators, computed from their independently assigned binary labels before any discussion to resolve disagreements. Agreement is high on all three labels (κ between 0.82 and 0.89), indicating consistent application of the annotation rubric prior to consensus discussion.

A.8 Sentry Implementation Details

A.8.1 Failure category and retrieval label definitions

In this section, we provide the detailed definitions and examples for the major failure categories as well as the retrieval labels.

Table 19: Inter-annotator agreement before consensus discussion for the human audit.

Label	N	**Percent agreement**	**Cohen's κ**
Intervention needed	200	94.5	0.89
Recovered	99	90.9	0.82
Helpful	113	95.6	0.82

Action validity failures. Action Validity Failures occur when the agent's action is not executable under the environment schema. These failures include malformed tool calls, missing arguments, invalid parameter types, unsupported tools, and invalid targets or controls.

Progress failures. Progress Failures occur when the agent's recent trajectory does not move the task toward completion. These failures indicate that the agent is stuck, repeating unproductive behavior, exploring without making useful progress, or failing to terminate at the right time. We use the following four retrieval labels:

- **Repetition or looping.** The agent repeats identical reasoning steps, tool calls, queries, or actions without gaining new information.
- **Planning stalls.** The agent recognizes the task objective but fails to convert it into a concrete next step. A typical example is that the agent generates plans but does not take useful actions.
- **Over-exploration.** The agent keeps gathering information even after it has enough evidence to make progress toward the task goal.
- **Termination miscalibration.** The agent stops too early before satisfying the task objective, or continues acting after the task is already effectively complete.

Reasoning & grounding failures. Reasoning & Grounding Failures occur when the agent's claims, plans, or actions are not supported by its observations. These failures indicate that the agent is acting on unsupported assumptions, drifting from the original objective, or choosing actions that do not follow from its reasoning. We use the following three retrieval labels:

- **Hallucination.** The agent introduces unsupported facts, assumptions, tool results, environment states, or task constraints that do not appear in the trajectory.
- **Objective drift.** The agent shifts away from the user's original objective.
- **Reasoning-action mismatch.** The agent's action does not follow from its own reasoning or from the current observation, such as choosing an action that contradicts its stated plan.

A.8.2 Retrieval and update rules

This appendix specifies the retrieval rule used by Sentry's external playbook. Each playbook entry is stored as a structured record containing a failure type, retrieval labels, a trigger pattern, a repair principle, and an insertion step. Sentry uses a deterministic label-filtered retrieval rule rather than embedding-based semantic search, making the retrieval behavior inspectable.

For a soft-repair failure, Sentry retrieves playbook entries in three steps. First, it keeps only entries with the same failure type and at least one retrieval label in common with the current failure. Second, it ranks the remaining entries by the number of overlapping labels, using recency only to break ties. Third, it inserts up to $k = 5$ entries into the soft-repair prompt. When more than five entries are available, Sentry greedily selects the highest-ranked entries, while limiting entries with the same primary retrieval label to at most two whenever possible. This diversity cap prevents guidance from being dominated by a single repeated failure subtype. If no playbook entry shares a retrieval label with the diagnosis, Sentry falls back to the k most recent entries of the same failure type.

The playbook is updated only after recovery verification. After a soft repair, Sentry waits H reasoning–action–observation cycles and invokes the verifier. If the local failure is judged as resolved, the

successful recovery is summarized into a new playbook entry. Although the physical playbook can grow over time, the inference-time guidance remains bounded by top-k retrieval, so the amount of injected guidance does not grow linearly with the playbook size.

A.8.3 SENTRY PROMPT TEMPLATES

This section provides the prompt skeletons used by Sentry. We show the shared structure rather than benchmark-specific schemas.

Sentry detector prompt skeleton

```
[SENTRY DETECTOR]
```

Role.
You are Sentry, an external runtime failure manager for an LLM task agent. Inspect the recent trajectory and decide whether the agent needs intervention. Do not judge final task success; judge whether the current local trajectory contains an execution-time failure.

Task objective.
{task_objective}

Recent trajectory window.
{recent_trajectory}

Required action schema.
{schema_description}

Failure taxonomy.
Action Validity: the proposed action is not executable under the environment schema.
Progress Failure: the recent trajectory repeats, stalls, over-explores, or terminates at the wrong time.
Reasoning & Grounding Failure: the agent introduces unsupported assumptions, drifts from the objective, or takes an action that does not follow from the reasoning or observation.

Output format.
Return whether intervention is needed, the failure type, retrieval labels, local evidence, and a concise diagnosis.

Sentry hard repair prompt skeleton

```
[SENTRY HARD REPAIR]
```

Detected issue.
The previous action is invalid or not executable under the required schema. Rewrite the action into one valid action that follows the same task objective and the current observation.

Task objective.
{task_objective}

Recent trajectory.
{recent_trajectory}

Invalid action.
{invalid_action}

Schema error.
{schema_error}

Required schema.
{schema_description}

Continuation instruction.
Output only the corrected action in the required action format.

Sentry soft repair prompt skeleton

[SENTRY SOFT REPAIR]

Detected failure type.
{failure_type}

Assigned retrieval labels.
{retrieval_labels}

General repair advice.
{failure_level_advice}
{label_specific_advice}

Why this was detected.
{local_evidence_bullets}

Detector signal.
{detector_diagnosis_and_constraint}

Insights from previous successful recoveries.
{top_k_retrieved_playbook_entries_or_empty_message}

Continuation instruction.
Continue the original task from the corrected next step. Keep following the original task protocol.

Required action format.
{schema_description}

Sentry escape verifier prompt skeleton

[SENTRY ESCAPE VERIFIER]

Role.
You are the Sentry recovery verifier. Your job is to judge whether the intervention fixed the exact failure that triggered the repair. Judge the specific retrieval labels and resolution criteria, not generic task success alone.

Task objective.
{task_objective}

Triggering failure.
{failure_type}
{retrieval_labels}
{trigger_evidence}

Recovery trajectory.
{post_intervention_trajectory}

Decision criteria.
A failure is resolved if the agent stops the diagnosed behavior and resumes valid, task-relevant, observation-grounded progress. A failure is not resolved if the same failure pattern persists, the agent drifts further, or the recovery introduces unsupported assumptions.

Output format.
Return resolved or unresolved, a short justification, and the evidence used for the judgment.

Sentry playbook update prompt skeleton

[SENTRY PLAYBOOK UPDATE]

Role.
You summarize only verified successful repairs into reusable playbook entries. Do not store generic reflections or advice that is not supported by the local trajectory.

Verified repair episode.
{failure_type}
{retrieval_labels}
{trigger_pattern}
{repair_guidance}
{post_repair_evidence}

Output format.
Return a concise playbook entry containing the failure type, subtype, trigger pattern, and repair principle. The entry should be specific enough to help a future similar failure, but not so narrow that it only applies to the original trajectory.

A.8.4 EXAMPLE PLAYBOOK

We show an example Sentry playbook from WebShop.

Progress failures.

- [rep-00001] **Repetition or looping.** If the same search query returns the same irrelevant result page, change one meaningful query variable from the user request rather than repeating the query.
- [rep-00002] **Repetition or looping.** If clicking an item and returning to search produces no new useful evidence, do not reopen the same item; inspect a different visible candidate or reformulate the query.
- [rep-00003] **Repetition or looping.** If navigation cycles between results, item page, and back-to-search without reward or new attributes, summarize what is known and take a state-changing action.
- [stall-00004] **Planning stall.** When the current page already exposes candidate items, convert the goal into one concrete action: inspect a candidate, choose an option, or buy if requirements are satisfied.
- [stall-00005] **Planning stall.** If the agent says it needs to check size, color, material, or price, the next action should be the specific click or search that can verify that attribute.
- [stall-00006] **Planning stall.** When no perfect result is obvious, choose the next most diagnostic action instead of continuing to restate the shopping goal.
- [explore-00007] **Over-exploration.** When a plausible candidate is visible, inspect it before issuing another broad search.

- [explore-00008] **Over-exploration.** When an item page already shows enough evidence for the requested constraints, commit instead of opening more unrelated candidates.
- [explore-00009] **Over-exploration.** Near the action budget limit, prefer narrowing or committing over another generic search.
- [explore-00010] **Over-exploration.** If several visible candidates share the requested category, compare those candidates directly instead of restarting from the search page.
- [term-00011] **Termination miscalibration.** If `Buy Now` is visible and the observed item satisfies the hard constraints, use `click[Buy Now]` instead of searching or navigating away.
- [term-00012] **Termination miscalibration.** If the task is not yet purchased, do not stop merely because a good candidate was found; finish with the valid purchase action.
- [term-00013] **Termination miscalibration.** If only one or two actions remain and a valid commit action is available, prioritize committing over further exploration.

Reasoning & grounding failures.

- [hall-00014] **Hallucination.** Do not infer size, color, material, or compatibility from the title alone when the observation has not shown that attribute.
- [hall-00015] **Hallucination.** If an attribute is missing from the item page, verify it through visible options or choose another candidate rather than claiming it is satisfied.
- [hall-00016] **Hallucination.** Do not claim "no matching item exists" after only checking one irrelevant result page.
- [drift-00017] **Objective drift.** Keep the original shopping constraints as hard filters; do not buy a nearby product just because it is cheap or generally similar.
- [drift-00018] **Objective drift.** If the current candidate matches the category but violates size, color, or price constraints, reject it and continue with the original objective.
- [drift-00019] **Objective drift.** Do not optimize for popularity, broad relevance, or low price when the user requested specific attributes.
- [mismatch-00020] **Reasoning-action mismatch.** If the reasoning says "check the item details," the action should inspect the candidate, not go back to search or buy immediately.
- [mismatch-00021] **Reasoning-action mismatch.** If the reasoning identifies one candidate as promising, the click target should match that candidate.
- [mismatch-00022] **Reasoning-action mismatch.** If the observation shows the required control, the next action should use that visible control rather than an unrelated search.